

# Byzantine Processors and Cuckoo Birds: Confining Maliciousness to the Outset

Danny Dolev (HUJI)\* and Eli Gafni (UCLA)<sup>†</sup>

## Abstract

Are there Byzantine Animals?

A Fooling Behavior is exhibited by the Cuckoo bird. It sneakily replaces some of the eggs of other species with its own. Lest the Cuckoo extinct itself by destroying its host, it self-limits its power: It does not replace too large a fraction of the eggs. Here, we show that any Byzantine Behavior that does not destroy the system it attacks, i.e. allows the system to solve an easy task like  $\epsilon$ -agreement, then its maliciousness can be confined to be the exact replica of the Cuckoo bird behavior: Undetectably replace an input of a processor and let the processor behave correctly thereafter with respect to the new input. In doing so we reduce the study of Byzantine behavior to fail-stop (benign) behavior with the Cuckoo caveat of a fraction of the inputs replaced. This goes beyond the reductions shown in the past that apply to colorless tasks. We establish a complete correspondence between the Byzantine and the Benign, modulo different thresholds, and replaced inputs.

This work is yet another step in a line of work unifying seemingly distinct distributed system models, dispelling the Myth that Distributed Computing is a plethora of distinct isolated models, each requiring its specialized tools and ideas in order to determine solvability of tasks (that is, Computability rather than Complexity). Thus, hereafter, Byzantine Computability questions can be reduced to questions in the benign failure setting. But vice versa too. In the more structured settings of asynchronous benign failures and synchronous Byzantine failures, researchers investigated correlated faults. We show that the known results about correlated faults in the asynchronous benign setting can be imported verbatim to the asynchronous Byzantine setting. Finally, as in the benign case in which we have the property that a processor can output once its faulty behavior stops for long enough, we show this can be done in a similar manner in the Byzantine case. This necessitated the generalization of Reliable Broadcast to what we term Recoverable Reliable Broadcast.

---

\*Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. The Rachel and Selim Benin School of Computer Science and Engineering Edmond J. Safra Campus. This work was supported in part by the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. This research project was supported in part by The Israeli Center of Research Excellence (I-CORE) program, (Center No. 4/11). This work was supported in part by the Ministry of Education of China through its grant to Tsinghua University. Email: danny.dolev@mail.huji.ac.il.

<sup>†</sup>Eli Gafni was partially supported by Israel-USA BSF grant 20152316 and an NSF grant 20170416. This work was supported in part by the Ministry of Education of China through its grant to Tsinghua University. Email: gafnieli@gmail.com.

# 1 Introduction

“Deep Learning” systems are built. They work, and will probably be the underpinning of a new Industrial Revolution. It happens now and it proceeds lacking serious Theoretical Foundation. Theory, if developed, will have to play catch-up, explaining why the systems work the way they do. Distributed Computing history is not unlike the current state of Deep Learning. Distributed Systems have been built and changed the world. Theory played and still plays catch-up. But does it play successfully? The hallmark of Theory is Unification. Reducing a seemingly chaos into few principles. This paper, motivated and helped by other recent successful unification steps, shows that Byzantine Models are, to paraphrase Ecclesiastes (Koheleth) [1], “Nothing New Under the Sun.”

And, it shows the value of Theory. This paper shows that the solvability of tasks in the Byzantine model can be reduced to a question about the solvability of tasks in the benign failure model. For a designer of a distributed system dealing with Byzantine processors, knowing whether the assumptions made theoretically allow for a solution, is as important as for a designer of a centralized system to know whether the system actually solves an  $NP$ -complete problem, or worse, and Undecidable one. The reduction we introduce simplifies answering this solvability question.

A limited reduction that applies mainly to the so called *colorless* tasks has been derived recently in [27] and then rediscovered in [14]. The authors of [27] claim their reduction is applicable conceptually to more than colorless tasks. In their later paper [26] there is no mention of [27], that with respect to the non-colorless task of Multi-Dimensional Epsilon-Agreement [25, 32], a task that we know now can be reduced to benign failures according to our Cuckoo model, they could have treated it in the benign model instead of resorting to analysis in the Byzantine model, with all the clutter it carries. The Cuckoo model, by doing away with the notion of “byzantine processors” removes barriers that are creating blind spots in our understanding. We give an explicit mapping of the Byzantine in terms of the Benign. The era of Byzantine Computability is over!

This investigation was motivated by recent unification steps that gave rise to our inquest as to their ramifications to the Byzantine model.

In [5], it was shown that as far as solving tasks, asynchronous models can be reduced to synchronous models with mobile-faults. If this is the case in the benign setting, what about the Byzantine? Also, there [5] processors are always “correct,” they do not “crash.” Their seemingly faulty behavior is due to the communication subsystem that is faulty. If we view Byzantine behavior not as a pitch-fork equipped Lucifer which takes over the soul of a processor, but just takes over a processor’s outgoing communication links, then there is meaning to what Byzantine processor outputs, it is a victim rather than the perpetrator.

Of course the Byzantine Model [29, 24] came to capture a processor malfunction. The point is that a theoretical model should be faithful in producing results commensurate with observable behavior. It should not necessarily try to capture the precise mechanism by which this behavior occurs in reality. In the benign faults model we try to capture “crash,” yet the right way to look at it is just as a worry about crash that therefore can be captured by communication delay, and thus allows a processor to rejoin after suffering the delay. Similarly, a Byzantine processor suffers malicious communication tampering, but once the tampering ceases the processor should be able to join in. Thus, we actually generalize the Byzantine Model of almost four decades, streamlining it with the benign fault mechanism modeling, while maintaining all legacy results.

The other motivating recent result [20, 15] raised the idea, in the benign case, that the connection between a processor and the thread it is supposed to execute, is more tenuous than was thought. The connection can be confined to the processor providing an input to the thread, and then fetching the output, obtained by the thread termination, to be delivered to the user. Between providing input and delivering an output to

the user, advancing the program-counters of active-threads - those whose input has been provided and whose execution has not been terminated, is a communal effort. It is a negotiation between all processors what a specific thread has “read” as the system state.

What about the Byzantine model then? If advancing program-counters is a communal effort, what meaning there is to a “misbehaving-thread?” All processors are responsible for it. Not surprisingly, the Cuckoo model we derive confines Lucifer only to the point of telling the system what input some of the threads should be run with. Although motivated by [20, 15] we end up associating a processor with executing its thread and the communal effort is about accepting or rejecting a proposed step of a processor. It turned out to be less involved than our initial “pure” communal effort.

In an initial effort ([18]) we got to provide reduction that applies for the “correct” processors, i.e. those whose communication links were never taken over by a Byzantine adversary. But how do we proceed once those correct processors output, if we now assume that the Byzantine Adversary gradually relinquishes its hold of some of the processors it interfered with before. How can a processor made whole (albeit with respect to its possible fake input) after other processors might have recognized it is faulty and stopped listening to it?

Current classical technique of “Reliable Broadcast” ([11]) does not allow recovery from such an implicit shut-off. Yet, in the benign failure “recovery” is trivial, as a seemingly “crashed” processor comes alive. Can it be that this fiat is doomed in the Byzantine case? This will contradict our sense of elegance that there should be *complete* analogy between the Byzantine and the Benign systems. Indeed, the most time-consuming part of the technical research involved in this paper was to eventually abandon the belief and the trials at a proof that this incomplete analogy is inherent and actually come with a modification to Reliable Broadcast (Recoverable Reliable Broadcast) that makes the analogy complete.

It also necessitated our generalization of the notion of what a model is while leaving it compatible with the past definitions. Traditionally, a model prescribes progress conditions independent of the execution of an algorithm. Here we allow progress conditions to change as a function of the outputs delivered. We assume a model in which once  $n - t$  outputs are delivered then at least from there on one additional processor that has not outputted will not experience further attacks on its communication anymore.

Finally, if our reduction is complete it should go the other way around too. We should be able to import models and results from the Benign to the Byzantine. In the synchronous and then the asynchronous benign models a body of work analyzed Correlated Faults (Also called Adversary Model) [23, 6, 16, 22, 21], and characterized the computability of such models. Our reduction allows to do same for the Asynchronous Byzantine once we have derived Reliable Broadcast [11] for Correlated Faults. We do just that. That said, we nevertheless keep the paper cast in the  $t$ -resilience and faulty/correct processors nomenclature, as after almost four decades most of us are more comfortable with it, even though it is exactly what this paper advocates to forget.

In the next section we elaborate on the models we deal with in this paper. Then embark on a sequence of reductions, introduce the Recoverable Reliable Broadcast, present Reliable Broadcast for the correlated faults model, review related work, and close with concluding remarks. But first, we summarize our contributions.

## 1.1 Contributions

### 1.1.1 Results

1. The  $t$ -resilient Asynchronous Byzantine model is equivalent to the  $t$ -resilient Asynchronous Benign model, where at most  $t$  of the user's inputs have been changed (i.e. the asynchronous Cuckoo model).
2. The  $t$ -resilient Asynchronous Byzantine model is equivalent to the a *synchronous*  $t$ -mobile model, where at most  $t$  of the user's inputs have been changed (i.e. the synchronous Cuckoo model).
3. Extend Correlated Faults to the Asynchronous Byzantine model.
4. Design a Recoverable Reliable Broadcast algorithm where a processor under attack can recover once some attacks stop (signaled by the number of outputs).

### 1.1.2 Concepts

1. Byzantine models as an attack on the communication subsystem rather than at attack on the processor's program.
2. A model notion as conditions for progress where the condition might change as a function of the outputs already delivered.
3. Ascribe meaning of output to a "byzantine processor," and require its progress once the attack stops.

## 2 Tasks, Models and Problem Statement

In this section we first repeat the notion of task, define the models we deal with, and state the problem this paper solves.

### 2.1 Tasks

A task is the elementary problem a distributed system implements. In distributed computing a task play the role a function plays in centralized computing. The task is a mathematical triple  $(I, O, \Gamma)$ , where  $I$  consists of set of tuples of sets of processors with their inputs,  $O$  consists of set tuples of sets of processors with outputs, and  $\Gamma$  is a set of binary relations assigning a set of output tuples from  $O$ , to any given input tuple from  $I$ . This relation is subject to the constraint the the set of processors in the input tuple, and the set of processors in the output tuples match. Hence, notions like "correct" and "faulty" that might come from a model, should not be referred to in a definition of a task.

The task prescribes to each participating set of processors with their inputs, what are the output combinations that they are allowed.

The notion of which processors participate in a protocol and which are not is model dependent. In general, if a set is participating it should be a set of processors that is closed under the relation "affected by." A participating processor should be oblivious to a non-participating one.

For instance, in the task of leader election on two processors  $p_0$  and  $p_1$ , the input sets are

$\{(p_0, p_0)\}, \{(p_1, p_1)\}, \{(p_0, p_0), (p_1, p_1)\}$ ,

the output sets are

$\{(p_0, p_0)\}, \{(p_1, p_1)\}, \{(p_0, p_0), (p_1, p_0)\}, \{(p_0, p_1), (p_1, p_1)\}$ ,

and  $\Gamma$  is

$\{((p_0, p_0), (p_0, p_0)), (p_1, p_1), (p_1, p_1)), (((p_0, p_0), (p_1, p_1)), ((p_0, p_0), (p_1, p_0))), (((p_0, p_0), (p_1, p_1)), ((p_0, p_1), (p_1, p_1)))\}$ .

## 2.2 Models

A model is a setting in which processors learn about each other's input, to produce each an output. The problem processors face is that they are required to output, under a limited knowledge of other processors input, and correspondingly, limited knowledge what other inputs other processors know. Hence to learn enough information to output, they are involved in negotiations of telling each other what input they know and updating their knowledge about others. Of course, they cannot negotiate forever and need to output at some point, hence the problem.

Depending on the assumptions about the means of the negotiations, some models are more “tightly coupled” than others allowing to solve a larger set of tasks. In this paper we do not consider systems that cannot solve  $\epsilon$ -agreement [17]. In this task we are given a finite interval of  $R$  of size at least  $1/\epsilon$ . Processors' inputs are arbitrary integers in the interval and all have to output integers within the convex hull of the inputs such that pairwise they output either the same integer or distinct integers which are adjacent. We do not consider models which cannot solve  $\epsilon$ -agreement, as we expect any reasonable system to have enough coordination power to solve this task.

Processors negotiate and output. After outputting they linger around to help other processors communicate (essentially by serving in effect as repeaters).

We take the view of a model following [2] as a contract between a specifier and a programmer (cf. Rely-Guarantee in [2]). Progress, in the form of additional processors outputting, is required in runs of the model where the specifier guarantees hold. We insert a new element to the contract not used in the past. The specification may be parametrized by processors' outputs. E.g. in the  $t$ -resilient case the specifier promises that all processors but  $t$  will participate and will infinitely often take steps in the negotiations (be alive). Under these conditions, the programmer is to deliver at least all but  $t$  outputs. Thereafter the specifier requires an output if at least all but  $t$  processors are alive, and at least among the live processors, there is a processor that hasn't outputted yet.

### 2.2.1 The Models in this Paper

We assume a set of  $n$  processors  $\Pi = \{p_1, p_2, \dots, p_n\}$ . The asynchronous settings is the classical asynchronous processors communicating over a point to point complete network. But, our definition of Asynchrony and Byzantine is not Classical. While traditionally faulty behavior has been ascribed to processors, we ascribe it to the communication subsystem. We assume an adversary that can manipulate outgoing messages. What the Byzantine and the Benign here have in common is that each has a set of bad-set of processors  $\mathcal{B}$ , where for all  $B \in \mathcal{B}$ ,  $B \subset \Pi$  and if  $B' \subset B$  then  $B' \in \mathcal{B}$ , i.e the set of bad-sets is subset closed. A good-set is any set  $\Pi \setminus B$ . We denote the set of good-sets  $\mathcal{G}$ . In the Benign Fault model the Adversary can remove messages sent by a set of processors from  $B$ . In the Byzantine case it can not only remove messages from  $B$  but also alter their content before delivery (but, it cannot generate messages on its own). Traditionally,  $t$ -resilience stands for a bad-set collection  $\mathcal{B}$  that contains any set of cardinality less than  $t + 1$ .

1. **Benign Asynchronous System:** The adversary can choose a set of processors of cardinality of at most  $t$  and remove some or all of their messages. Otherwise it can delay any message sent by anybody for any finite time. This allows the programmer to use the construct “wait until received messages from a (good-)set of cardinality  $n - t$ ,” when all processor are programmed to send messages to all.

Under this conditions, the programmer is obliged to deliver at least  $n - t$  outputs. From there on

the delivery of outputs hinges on the additional condition that the messages sent by some processor which hasn't outputted will not be delayed infinitely long. (Traditionally, some researchers allowed processors which have output to halt. This is a mistake and cannot be sustained in the Byzantine Asynchronous Model. Therefore, in general, we take the view that processors never halt.)

2. **Byzantine Asynchronous system:** In this system in addition to the above, the adversary has the power to change the content of messages sent from a fixed set processors of cardinality at most  $t$ . Notice that we do not allow the adversary to inject messages into the system by itself. (This last condition is restrictive with respect to the view that Byzantine means that a processor was taken over, yet we claim that nevertheless all legacy results hold.)

**Remark** The traditional threshold Byzantine Asynchronous model motivated a variation of the  $\epsilon$ -agreement task in which the at most two integers output have to be in the convex-hull of every  $n - t$  inputs [17]. It is easy to see that the new variation we may pose in which we require the output integers to be in the convex-hull of every good set, is also solvable using the same techniques as in [17]. And, in the multi-dimensional version of the problem we can replace the  $t$  “bad” processors by an adversary that captures  $t$  (i.e.  $t, n/4$  replace by good sets each 4 of which have nonempty intersection). This allow to calculate the resulting set consensus power via reduction to the benign adversary.

*The Cuckoo Asynchronous system:* This system is a Benign Asynchronous system in which the adversary not only remove messages from a set of cardinality at most  $t$  but it also might, unbeknownethed to the affected processors, change their inputs.

The *problem statement* of this paper is to show the equivalence between the Byzantine Asynchronous system and the Cuckoo Asynchronous system.

Our notion of equivalence between two models is that they both solve the same set of tasks. In the Byzantine/Cuckoo model, it is up to the user to define this notion of solvability since the adversary might produce an input combination not expected by the user. We claim equivalence whichever way the user will resolve such a notion, as our mechanism of showing the equivalence is by showing that any protocol in one model can be emulated by a protocol in the other.

**Remark:** Once we are in the Byzantine case past  $n - t$  outputs we tread in an uncharted territory. Never before the concept of an output from a “byzantine processor” was considered. What we try to capture is a correct processor, that snoops at others communication while what it sends may be under attack. And then, whether it can make itself whole once the attack ceases (albeit with possibly changed input). When does an attack cease? What the specifier promises the programmer, in this case, is that once  $n - t$  outputs are delivered, one of the processors that have not output yet will cease to be under attack. From there on its messages would not be altered any more and all its messages sent thereafter will eventually be delivered. We use the term “become correct” for such a processor, and “correct” for a processor whose messages are never attacked by the adversary. Observe that after a processor becomes correct, the adversary can still choose to deliver old and manipulated messages the processor had sent prior to recovering (becoming correct).

3. **Synchronous Mobile Adversary:** We show that the two asynchronous systems above are equivalent to synchronous systems. The Benign Asynchronous is equivalent to a synchronous system in which the choice of at most  $t$  processors from which the adversary may remove messages is a choice *per round*, hence the name “mobile” adversary. We also show that the Byzantine Asynchronous system is equivalent to synchronous system as above, only that now the adversary can also tamper with

messages rather than just remove them. The point of the paper is that this additional power of tampering can be confined to appear just as changing some  $t$  inputs at most, and from there on the adversary can just remove messages, i.e resulting in the Synchronous Mobile Cuckoo Model. We denote these systems as BI-MO( $B_{initial}$ ) where in the benign asynchronous case the input set to the synchronous equivalent system is the empty set.

**Remark:** The gap between the Asynchronous system we presented and the Synchronous Mobile is large. In the latter, in a round, the missed messages are all confined to messages from a single set of at the most cardinality  $t$ . In the former, the asynchronous case, each processor might miss at most  $t$  messages too, but the missed messages are not confined to come from the same set of cardinality at most  $t$ . The direct synchronous analog to the asynchronous system is to allow the adversary to attack the reception process rather than the transmission process: For every process  $p_i$  in a round, the adversary can choose at most  $t$  incoming messages to be removed. This leads us to the synchronous system denoted by BI-Synch. The BI-Synch will be mediating between the Asynchronous and the Synchronous breaking the final reduction into two intermediate reductions.

### 2.3 The Correlated-Faults Adversary Model

In [23, 6] it was suggested to replace the condition for synchronous Byzantine agreement from  $n \geq 3t+1$  to a list of faulty set  $\mathcal{B}$  under the condition that no union of three sets from  $\mathcal{B}$  is  $\Pi$ . By showing that this condition is enough to implement Reliable Broadcast all our results cast in the resilience model carry verbatim to the correlated case with all the implications and results of correlated case in the Benign model (see Section 6). Obviously if the condition is violated and the union of some three sets is  $\Pi$ , then the Adversary can cause network partition with no possibility of a meaningful computation.

## 3 Simulating a protocol running in BI-Synch system

In this part we emulate any protocol running in a BI-Synch system by a protocol running in a  $t$ -Byzantine resilient asynchronous system. Thus, we prove that the adversary in an asynchronous system with the  $t$  processors it controls and with the ability of delaying messages to all processors does not have any more power than the adversary in the synchronously running BI-Synch system.

The idea behind the simulation is to completely simulate the original protocol running in a BI-Synch system by a protocol running at each processor in a  $t$ -Byzantine resilient asynchronous system, such that every processor that becomes correct in the  $t$ -Byzantine resilient asynchronous system would get the same output as it would have gotten in the BI-Synch system. We assume that initially there are  $n - t$  correct processors, and new processors are relieved from the adversary's control and become correct during the run of the protocol.

In the emulation appearing in this section we make use of two elements. The first element employed is to make sure that every processor commits to the message it sends in each round in a way that if any processor accepts a message  $m$ , everyone will eventually accept the same message  $m$ , and before accepting  $m$  it accepts all  $m$ 's causally ordered prior messages. Thus, the first element, CO\_SEND (Algorithm 2), is a *Causally Ordered Recoverable Reliable Broadcast primitive*, which uses a generalization of the classical reliable broadcast as a building block, combined with the causality of message delivery (generalizing a combination of [11, 7]). As we detail below, to further limit the ability of the adversary to confuse processors we instruct each processor, after sending its initial input, to send only the list of IDs of the processors



---

**Algorithm 1:** Simulating a BI-Synch system deterministic protocol  
in an asynchronous system with  $n > 3t$ .

---

```

1.  set  $\forall k \text{ } accept[k] := \emptyset;$  /* executed at processor  $p$  */
2.  set  $\mathcal{M} := \emptyset; \bar{\mathcal{M}} := \emptyset;$  /* the sets of senders whose messages were processes at the given rounds */
3.  set  $O := \emptyset;$  /* the set of accepted messages that were not processed yet, and the set of processed messages */
4.  /* the set of processors that their SM reached an output at the processor */
5.   $r := 1;$  /* the round number */
6.  invoke  $CO\_SEND(r, p)$  to broadcast  $\mathcal{I};$  /* broadcast the input value, a processor sends also to itself */
7.  do until  $SM_p$  halts: /* participate in all  $CO\_SEND(\ell, *), \ell \leq r$ , protocols */
8.  wait until  $|accept[r]| \geq n - t$  and  $p \in accept[r];$ 
9.  /* "empty line" - for later use */
10.  $r := r + 1;$ 
11. invoke  $CO\_SEND(r, p)$  to broadcast  $accept[r - 1];$  /* broadcast the accepted and processed set in round  $r - 1$  */
12. end.

```

**In the Background:**

```

12. Execute for each  $\langle r', p_i, \pi \rangle \in \mathcal{M};$  /* accepted message from  $p_i$  for round  $r'$  */
13. if  $r' = 1$  then start  $SM_i$  with input  $\pi;$  /* start a SM with the initial input */
14. if  $r' > 1$  then
15.   let  $M := \{m_j \mid p_j \in \pi \text{ and } SM_j[r' - 1] \text{ sends } m_j \text{ to } p_i\};$  /* the values  $p_i$  should have received */
16.    $SM_i[r'] := \mathcal{F}(M, SM_i[r' - 1], r');$  /* apply protocol  $\mathcal{F}$  to determine the next state of  $SM_i$  */
17.   if  $SM_i$  outputs then  $O := O \cup \{p_i\};$  /* Denote the outputting processor */
18.    $\mathcal{M} := \mathcal{M} \setminus \langle r', p_i, \pi \rangle;$ 
19.    $\bar{\mathcal{M}} := \bar{\mathcal{M}} \cup \langle r', p_i \rangle;$ 
20.    $accept[r'] := accept[r'] \cup \{p_i\}.$ 

```

---

it claims to accept and process messages from in the previous round, without any additional content (we differentiate between receiving a message, agreeing to accept it as a legitimate message to be processed, and processing it). The second element is to locally simulate the state of the original protocol at every other processor, according to the list of processors' IDs being accepted, to determine what the original (simulated) protocol instructs each processor to do, what values should be sent and which should be received.

Let  $P$  be a deterministic message passing protocol that is executed in a BI-Synch system. We will show a simulation of  $P$  in a  $t$ -Byzantine resilient asynchronous system. In the simulation, in the first round each processor,  $p$ , uses  $CO\_SEND(1, p)$  to broadcast its own input value,  $\mathcal{I}$ . In each subsequent round,  $r$ , each processor,  $p$ , uses  $CO\_SEND(r, p)$  to broadcast to everyone a set,  $\pi_{r-1}$ , of the IDs of the processors from which it accepted and processed messages in the previous round, round  $r - 1$ .

We start with an overview of the simulation protocol (Algorithm 1). Each processor,  $p$ , maintains locally  $n = |\Pi|$  state machines,  $SM_i$ ,  $1 \leq i \leq n$ , to reflect the state machines of the original (simulated) protocol at each other processor. When  $p$  accepts via  $CO\_SEND$  a message  $\langle 1, \mathcal{I} \rangle$  from a processor  $p_i$ , it initiates state machine  $SM_i$  with the input value  $\mathcal{I}$ . If  $p$  does not accept such a message from a processor, say  $q$ , it does not start the state machine  $SM_q$ . The properties of  $CO\_SEND$  ensure that if any processor will ever accept a message  $\langle 1, \mathcal{I} \rangle$  from a processor  $q$ , then eventually processor  $p$  will also accept it too.

When processor  $p$  accepts via  $CO\_SEND$  a message  $\langle 2, \pi_1 \rangle$  from a processor  $p_i$ , it uses the initial input values of all  $q_j \in \pi_1$  as the set of input values to  $SM_i$  to determine what values the simulated protocol  $P$  instructs processor  $p_i$  to send to every other processor in round 2 of protocol  $P$ . Since  $CO\_SEND$  implements Causally Ordered Reliable Broadcast, before processor  $p$  processes  $\langle 2, \pi_1 \rangle$  from a processor  $p_i$ , it already



accepted and processed all  $\langle 1, \mathcal{I} \rangle$  messages from all processors in  $\pi_1$ , and therefore knows the values  $p_i$  should have received from the members of  $\pi_i$ . Observe that if the adversary causes the message from processor  $p_i$  to claim to accept an input from a processor, say  $q$ , that did not send it an input, then the  $\langle 2, \pi_1 \rangle$  message received from  $p_i$  will be put aside and the CO\_SEND invoked by  $p_i$  will not be completed at any other processor until (and if) the input from  $q$  will be processed.

Now recursively, when processor  $p$  accepts via CO\_SEND a message  $\langle r, \pi_{r-1} \rangle$  from a processor  $p_i$ , it uses the values every  $q_j \in \pi_{r-1}$  should have sent in round  $r - 1$  of protocol  $P$  to  $p_i$  according to  $q_j$ 's state machine  $SM_j$  at round  $r - 1$ , as values received by  $SM_i$  in the previous round (round  $r - 1$ ) to determine what values the simulated protocol  $P$  instructs processor  $p_i$  to send to every other processor in round  $r$ .

The simulation protocol, presented in [Algorithm 1](#), maintains four data structures,  $\mathcal{M}$ ,  $\bar{\mathcal{M}}$ , *accept* and  $O$ . The set  $\mathcal{M}$  contains the messages that were accepted via CO\_SEND and were not processed yet. There is at most one such message per sender (because all messages of the same sender are casually related). Each entry in  $\mathcal{M}$  contains a round number, say  $r$ , a processor ID, say  $p_i$ , and the set of processors' IDs, from which processor  $p_i$  claims to have accepted and processed messages in round  $r - 1$ .

The set  $\bar{\mathcal{M}}$  contains the list of processors whose messages were already processed by processor  $p$ , and their SMs are updated accordingly. Each entry in  $\bar{\mathcal{M}}$  contains a round number, say  $r$ , a processor ID, say  $p_i$ , indicating that round  $r$  message from  $p_i$  was accepted and processed. By the CO\_SEND properties, every processor that processes a round  $r$  message from  $p_i$  processes the identical message.

The CO\_SEND protocol, [Algorithm 2](#), is a communication procedure that exchanges messages and holds received messages until they are ready to be accepted and processed. The CO\_SEND properties (as we later explain) imply that when an entry is added to  $\mathcal{M}$ , all casually prior entries were already accepted and processed (thus, are already in  $\bar{\mathcal{M}}$ ), and as such are reflected in the respective state machines (as we explain later). Therefore, each message in  $\mathcal{M}$  can be processed independently, since there are no causal dependencies among them. Processing a message is just applying it to the state machine of the sending processor, using the current state of the state machines of all the processors it claimed to accept their messages in the previous round. Once a message is processed it is removed from  $\mathcal{M}$  and added to  $\bar{\mathcal{M}}$ . Observe that the simulation may indicate that at a certain round some processor is not sending a value to some other processor, then in such a case no such value is produced as an input to the relevant state machine.

The third data structure (*accept*) is the set of processors whose messages were accepted and processed in the given round. *accept* $[r]$  is the current set of all the processors whose round  $r$  messages were accepted and processed by  $p$  via CO\_SEND. Once  $|accept[r]| \geq n - t$ , and  $p \in accept[r]$ , processor  $p$  uses CO\_SEND( $r + 1, p$ ) to broadcast the set *accept* $[r]$ . After broadcasting this message processor  $p$  continues to accept all rounds' messages via CO\_SEND and continues to apply them to the various state machines. Each processor continues this process, outputs its output, and sends messages until its state machine halts.<sup>1</sup>

The fourth data structure is the list of processors whose state machines produced outputs at the processor. Whenever the processor learns that some state machine produced an output it adds it to the set  $O$ .

Observe that the emulation, presented in [Algorithm 1](#), produces per each processor an agreed upon sequence of sets of values  $M_0, \dots, M_{r-1}$  processed by its SM in the related rounds, thus, simulating the exact behavior of protocol  $P$ . This implies that the above emulation is a protocol to simulate in a  $t$ -Byzantine resilient asynchronous system a deterministic message passing protocol,  $P$ , in a BI-Synch system.

The enabling properties of the simulation reside in the details of CO\_SEND, which we now describe. The CO\_SEND protocol, [Algorithm 2](#), is invoked per processor per sending round and consists of 3 conceptual

<sup>1</sup>The processor continues to participate in the CO\_SEND protocols of other processors even after it halts. One can add a halting task that enables a processor to halt once enough other processors reach the right stage.

---

**Algorithm 2:** CO\_SEND( $r, s$ ): A casually ordered reliable broadcast  
with asynchronous system with  $n > 3t$ .

---

$\mathcal{M}, \bar{\mathcal{M}}$  and  $O$  are globally maintained sets */\* executed by processor  $p$  with sender  $s$  in round  $r$ , invoked once per round \*/*

**Sender's Protocol:**

*/\*  $s$  is the sender and  $v_s$  the value it broadcasts \*/*

1. The sender  $s$  invokes RecRB( $s$ ) with  $v_s$  to send to all. */\* RecRB is a recoverable Reliable Broadcast (see Section 4) \*/*

**Any Processor's Protocol:**

2. **Upon receiving** an RecRB( $s$ ) protocol message with value  $v$ :
  3.     **if** ( $r > 1$ ) **wait until**  $s \in v$  **and**  $\forall q \in v, \langle r-1, q \rangle \in \bar{\mathcal{M}}$ ; */\* wait for the causally prior messages \*/*
  4.     join RecRB( $s$ ) as participant; */\* joined at most once per sender per round \*/*
  5.     **case** accepted RecRB( $s$ ) with value  $v'$ :
  6.     **if** ( $r > 1$ ) **wait until**  $\forall q \in v', \langle r-1, q \rangle \in \bar{\mathcal{M}}$ ; */\* wait for the causally prior messages \*/*
  7.     **Accept:**  $\mathcal{M} := \mathcal{M} \cup \langle r, s, v' \rangle$ . */\* accept message  $v'$  as the message sent by processor  $s$  in round  $r$  \*/*
  - Case** A new processor is added to  $O$ , i.e., outputs: */\* Try to recover \*/*
  8.     **if**  $|O| \geq n - t$  **then** sender  $s$  repeats the last send of Line 1. */\* repeat the last sending \*/*
- 

parts. In the first part the sender of the current instance of the protocol (in Line 1) invokes RecRB, a recoverable Reliable Broadcast (see Section 4), to send its value to everyone.

RecRB ensures that:

- RRB1 If the sender  $s$  is correct when it invokes RecRB with value  $v$ , then every processor will eventually accept RecRB( $s$ ) with the value  $v$ .
- RRB2 If any processor accepts RecRB( $s$ ) with a value  $v'$ , then every processor will accept it with the same value.
- RRB3 If sender  $s$  did not invoke RecRB( $s$ ) no processor will accept RecRB( $s$ ).
- RRB4 If the sender  $s$  becomes correct after invoking the protocol, then every processor will eventually accept RecRB( $s$ ).

The second part is executed by every processor. In the first round, every processor joins the RecRB invoked by the sender. In later rounds processors validate the value, since the value is a set of processors whose messages were accepted in the previous round. We need to ensure that if the RecRB will be accepted, the resulting value will be consistent with the local views of all processors. To ensure that, if a processor learns that the RecRB was invoked by  $s$  it waits until the value sent is contained in its own view of the previous round (Line 3). It joins the invoked RecRB only when this holds. It joins a single invocation per sender per round. The local testing ensures that the accepted value will eventually enable every processor to apply the accepted value to the local state machine of  $SM_s$ . Since the accepted value may differ from the value first seen by the processor, the participating processor waits again (Line 7) until its local view becomes consistent with the accepted value. This is guaranteed to happen eventually, since the accepted value tested by any processor will eventually hold at any other processor.

If  $s$  is not correct when it invokes the protocol, the protocol may get blocked. For example, if the local test in (Line 3) fails at all processors. If  $s$  is recovered (becomes correct) while running this protocol, when it resends its value in (Line 1) the protocol recovers. The third part intends to catch exactly that. Every time a new processor recovers while running CO\_SEND, its CO\_SEND will complete successfully. The adversary may cause each processor it controls to resend values at most  $2t$  times.

Given the above discussion, it is clear that if the sender is correct, all processors will eventually complete the protocol and will accept its value. Moreover, if any processor accepts a message, every processor will end up eventually accepting the same message, after accepting and processing all causally prior messages to that message. Thus, we outlined the proof of the following claim.

**Lemma 1.** *If  $n > 3t$ , then [Algorithm 2](#) implements a Causally Ordered Recoverable Reliable Broadcast transport layer in which if a sender  $s$  uses CO\_SEND to send its messages, each processor accepts messages that satisfy:*

- CO1: *If a correct sender,  $s$ , sends a consecutive sequence of messages, then every processor accepts the sequence in the same order that it was sent.*
- CO2: *For  $r > 1$ , if a processor,  $p$ , accepts a  $v$  via CO\_SEND( $r, s$ ), then  $s \in v$  and  $p$  already accepted  $v_j$  via CO\_SEND( $r - 1, p_j$ ), for every  $p_j \in v$ .*
- CO3: *If a processor,  $p$ , accepts a  $v$  via CO\_SEND( $r, s$ ), then every processor will accept  $v$  via CO\_SEND( $r, s$ ).*

The above discussions, [Lemma 1](#), and given that a processor moves to the next round, once it accepts and processes current round messages from a set of  $n - t$  processors, imply the following result.

**Lemma 2.** *Given a deterministic protocol  $P$  that is viewed as a function  $\mathcal{F}(M_{r-1}, S_p, r)$ , for  $r \geq 1$ , in a BI-Synch system, the emulation protocol presented in [Algorithm 1](#) simulates it at every processor in a  $t$ -Byzantine resilient asynchronous system, provided that  $n > 3t$ .*

*Proof.* To prove the claim what we need to show is a mapping of processors and inputs from a  $t$ -Byzantine resilient asynchronous system to the corresponding processors and inputs in a BI-Synch system; and show that every run in the  $t$ -Byzantine resilient asynchronous system corresponds to a possible run in the BI-Synch system and that correct processors in the  $t$ -Byzantine resilient asynchronous system obtain the same output the corresponding processors obtain in the BI-Synch system.

Let  $B$  be the set of Byzantine processors the adversary initially controls. Let  $G = \Pi \setminus B$  be the appropriate set of correct processors. We map the processors' IDs the same in both systems, and we will consider those processors controlled by the adversary in the  $t$ -Byzantine resilient asynchronous system as the Cuckoo set, those receiving a bad input in the BI-Synch system.

Assign to a processor in the BI-Synch model the input value a correct processor obtained from the respective CO\_SEND in [Line 5](#) of [Algorithm 1](#), when running it in the  $t$ -Byzantine resilient asynchronous system (processors that never take an action in the  $t$ -Byzantine resilient asynchronous system are part of  $B$  and will be considered as processors in the BI-Synch model whose messages never arrive to any other processor). Thus, at the beginning of the first phase, once the inputs are introduced, the state of the corresponding processors are the same.

The set of IDs each processor receives (and accepts) in the first round is the set it sends when executing [Line 10](#). The state of each processor  $p$  at the end of round 1 is the state it obtains when processing its round 1 message when executing [Line 16](#). Observe that every correct processor eventually complete the first round and send its initial input. Thus, at the end of round 1, the local state of processor  $p$  is the same state its state in BI-Synch model at the end of round 1. Observe that, by [Lemma 1](#), any other processor that computes locally the state of processor  $p$ , when it processes processor's  $p$  round 1 message when executing [Line 16](#) will obtain the same state at its local state machine of processor  $p$ .

Now, by induction, we can show that the set of processors each processor  $p$  accepts in round  $r$  is what it sends in [Line 10](#), contains a set of  $n - t$  processors. Its state after processing the round  $r$  messages (at the

beginning of round  $r + 1$ ) is what every processor obtains when processing its broadcasted message when executing [Line 16](#).

When the protocol in the BI-Synch model instructs a processor  $p$  to produce an output the SM at every processor will produce in the  $t$ -Byzantine resilient asynchronous system the identical output. Observe that in our model all processors will run all state machines of all initially correct processors, and thus will produce the outputs. Once a SM produces an output at any processor, it eventually produces the same output at every other processor. Once a processor learns about an output (executing [Line 17](#) of [Algorithm 1](#)) it updated its list of outputs. Thus, once all initially correct processors output, every processor learns about that, and will try to recover. One of them will recover and will reach an output state. The process repeats itself, and eventually all processors output on all state machines.  $\square$

## 4 Recoverable Reliable Broadcast

Traditional Reliable Broadcast ensures properties [RRB1] - [RRB3], below. The adversary may cause a traditional Reliable Broadcast protocol to block when it is invoked by a processors it controls, by sending conflicting values to different processors. Our aim is to ensure that once a processor recovers, the protocol it previously invoked while being controlled by the adversary will recover and complete. To obtain that we extend the properties of Reliable Broadcast to be:

- RRB1 If the sender  $s$  is correct when it invokes RecRB with value  $v$ , then every processor will eventually accept RecRB( $s$ ) with the value  $v$ .
- RRB2 If any processor accepts RecRB( $s$ ) with a value  $v'$ , then every processor will accept it with the same value.
- RRB3 If sender  $s$  did not invoke RecRB( $s$ ) no processor will accept RecRB( $s$ ).
- RRB4 If the sender  $s$  becomes correct after invoking the protocol, then every processor will eventually accept RecRB( $s$ ).

The Recoverable Reliable Broadcast protocol presented in this section ([Algorithm 3](#)) obtains these properties. The protocol uses as a building block the classical Reliable Broadcast protocol ([Algorithm 6](#), see [Section 6](#)), that was developed by Bracha [[11](#)], and was used by many researchers ever since.

The challenge is in ensuring that despite the degree of freedoms the adversary has in trying to confuse processors by changing the content of messages it can't block recovered processors from making progress. The high level idea of the protocol is that the sender will repeatedly push forward a value via a sequence of sending attempts, until any of the attempts collects support for the same value from at least  $n - t$  processors. Each attempt brings every processor to invoke a Reliable Broadcast to send the value it received from the sender for that attempt. Each participant maintains a data structure in which it collects the accepted Reliable Broadcasts and the values it accepted for each attempt.

The protocol is composed of three parts. In the first part the sender sends in each iteration a value and the set ( $H[k - 1]$ ) of accepted Reliable broadcasts of the previous iteration ([Line 5](#)). In the first iteration this set is the empty set. The processor waits ([Line 6](#)) until it accepts at least  $n - t$  Reliable Broadcasts. Once this happens it determines ([Line 7](#)) the value it should send in the next attempt. The value chosen is either a unique value appearing at least  $t + 1$  times or the original value the sender started the protocol with. Notice that if the sender was correct when it invoked the protocol the value is always the value it started with. The sender repeats this process until for any attempt ([Line 8](#)) there are  $n - t$  identical values. In such a case it

---

**Algorithm 3:** RecRB( $s$ ): Recoverable Reliable Broadcast

---

with asynchronous system that satisfies the Byzantine Fault predicate

$O$  is a globally maintained sets

*/\* executed by processor  $p$  with sender  $s$ , invoked once per round per sender \*/*

1. **set**  $H[k] = \emptyset$  for any  $k \geq 0$

*/\*  $H[k]$  includes all accepted RB of round  $k$  \*/*

**Sender's Protocol:**

2.  $k := 0; v := v_s;$

3. **repeat**

4.  $k := k + 1;$

5. **send**  $\langle v, k, H[k-1] \rangle$  to all;

*/\*  $s$  is the sender and  $v$  the value it attempts to broadcast \*/*

6. **wait until**  $|H[k]| \geq n - t$

*/\*  $n - t$  different RB were accepted \*/*

7. **if**  $\exists$  unique  $\bar{v}$ , s.t.  $|\{q \mid (q, \bar{v}) \in H[k]\}| \geq t + 1$  **then**  $v := \bar{v}$  **else**  $v := v_s$ ;

*/\* either  $\bar{v}$  or the input value \*/*

8. **until**  $\exists k', v'$ , s.t.  $|\{q \mid (q, v') \in H[k']\}| \geq n - t$

*/\*  $n - t$  different RB were accepted with a value  $v$  \*/*

**Any Processor's Protocol:**

*/\* the sender also run this protocol \*/*

9. **case received**  $\langle v, k, H'[k-1] \rangle$  from  $s$ :

10. **wait until**  $H'[k-1] \subset H[k-1]$

*/\* the sender also run this protocol \*/*

11. **if** (did not invoke RB for  $k$ ) and  $(\nexists v' \neq v$  s.t.  $|\{q \mid (q, v') \in H'[k-1]\}| \geq t + 1)$

12. **then** invoke RB( $k, p$ ) with value  $v$ ;

*/\* executed at most once per  $k$  \*/*

13. **case accepted**  $\bar{v}$  via RB( $k, q$ ):

14. add  $(q, \bar{v})$  to  $H[k]$ ;

15. **case**  $\exists k', v'$ , s.t.  $|\{q \mid (q, v') \in H[k']\}| \geq n - t$ :

*/\*  $n - t$  different RB were accepted with a value  $v$  \*/*

16. **accept** RecRB( $s$ ) with value  $v'$ .

*/\* accept message  $v'$  as the message sent by processor  $s$  via RecRB \*/*

**Case** A new processor is added to  $O$ , i.e., outputs:

*/\* Try to recover \*/*

17. **if**  $|O| \geq n - t$  **then** sender  $s$  repeats the last send of [Line 5](#).

*/\* repeat the last sending \*/*

---

can stop, and it knows that every processor will end up seeing that value in at least one iteration, as we argue below.

The second part is executed by every participant, including the sender. When a processor receives from the sender ([Line 9](#)) a value and a set of the accepted Reliable Broadcasts at the previous iteration, as claimed by the sender, it carries various validation tests prior to invoking the next Reliable Broadcast. It first waits until the set of Reliable Broadcasts it accepts ( $H[k-1]$ ) contains the set ( $H'[k-1]$ ) of the sender. Next it checks whether the value received from the sender does not contradict the set  $H'[k-1]$ . If both hold and the processor did not already invoke a Reliable Broadcast with index  $k$  it invokes one. The processor collects all accepted Reliable Broadcasts in its data-structure  $H[k]$ , whenever any is accepted, in the context of the current RecRB ([Line 13](#)), until it identifies that its data structure contains at least  $n - t$  identical values for some iteration ([Line 15](#)). Once this happens it accepts that value as the sender's value of the invoked RecRB. Because of recovering, a processor may get more than a single message per iteration (at most  $t$  such messages in total). It invokes the corresponding reliable broadcast on at most one of these messages.

Observe that if any processor accepts a value for the current RecRB, that value will be accepted by every processor. The reason is that the Reliable Broadcast properties imply that what one data structure contains, every data structure will eventually contain. Moreover, we argue that there will not be two different values accepted. Let  $\bar{k}$  be the minimal index for which any processor eventually collects at least  $n - t$  accepted Reliable Broadcasts for some value  $\bar{v}$ , then the value being sent by the sender in iteration  $\bar{k} + 1$  can contain only  $\bar{v}$ . If it differs from that value (due to adversarial manipulation), then it will fail the tests of [Line 10](#) and

Line 11 at any processor, and would never be sent by any processor.

The third part intends to help the sender to complete the protocol in the case it recovers and its previous message caused the participating processors not to invoke the corresponding Reliable Broadcasts. When it identifies that the number of processors that gave output is more than  $n - t$ , it knows that there is a chance it will be recovered; and in such a case it repeats the last sending each time it learns about another processor outputting.

Observe that if the sender was correct before the invocation of RecRB, then eventually every processor will see  $n - t$  identical copies of its value in the first iteration, or in any later iteration. Also notice, that if any processor accepts the RecRB of the sender, everyone will eventually accept it with the same value. Thus, proving the 4 properties of RecRB.

## 5 Simulating a protocol in a BI-MO( $B_{initial}$ ) system

To finalize the main result of the paper we will now expand the simulation from simulating a protocol  $P$  that runs in a BI-Synch system to a protocol in a BI-MO( $B_{initial}$ ) system. The extension is to ensure that before a processor adapts the *accept* set it communicates with others to converge to *accept* sets such that all sets have at least  $n - t$  processors in common. To achieve that we introduce another element, we run a couple of rounds of the equivalent to a full information message exchange to make sure that everyone shares messages from a set of at least  $n - t$  processors. Once this happens the processor takes its next step.

COMMONCORE, is an adaptation of the Get-Core approach mentioned in [7] (attributed to the second author) and a variation of it that was later presented in [3] as Binding Gather, and using ideas from [4].

Each processor invokes the COMMONCORE protocol, appearing in Algorithm 4, with a set of at least  $n - t$  different processors IDs. Each processor,  $p$ , returns as an output a set of at least  $n - t$  different processors' IDs, such that at least  $n - t$  of them are shared by the outputs of all processors. The COMMONCORE properties are:

- *Validity*: At each processor, the output set of IDs contains the input set of IDs.
- *Commonality*: There exists a set of  $n - t$  IDs that appears in the output set of every processor.
- *Termination*: All correct processors eventually output some non-empty set of IDs.

A set that is in every output set is called a common core. The COMMONCORE primitive is described in Algorithm 4. In the first round, everyone sends its *accept* $[r]$  set. In the background the processor continues to update its *accept* $[r]$  set with messages it continues to accept. To complete the first round of COMMONCORE, it waits to receive at least  $n - t$  sets that are contained in its current state of the set *accept* $[r]$ . This will eventually happen due to the CO\_SEND properties, and the fact that there are at least  $n - t$  correct processors. Once this happens it sends again its current set and waits again to receive at least  $n - t$  second round sets that are contained in its current state of the set.

Observe that a recovering processor does not need to repeat the last sending as it did in previous protocols..

The correctness proof is similar to the proof of get-core in [7]. The original proof did not include Byzantine nodes, therefore we need to change it a bit. Define a table  $T$  with  $n - t$  rows and  $n - t$  columns, that refer to a set  $G$  of  $n - t$  correct processors. The *accept* $[r]$  value of each correct processor contains at least  $n - t$  IDs, therefore it contains at least  $n - 2t \geq t + 1$  IDs of processors in  $G$  represented in  $T$ . For  $p_i, p_j \in G$ , entry  $T[i, j]$  in the table is 1 if  $p_j$  is one of the  $n - t$  processors that  $p_i$  waited for in order to



---

**Algorithm 4:** COMMONCORE( $accept[r]$ ), the Common Core protocol

---

*/\* the input set  $accept[r]$  is updated contineously in the background according to the messages accepted via CO\_SEND and processed in Algorithm 1 \*/*

1. **step 1**  $send(r, 1, accept[r])$  to all; */\* send the input set to all, a processor sends also to itself \*/*
  2. **wait until**  $|\{j \mid received(r, 1, \pi_j) \text{ from } p_j, \text{ and } \pi_j \subseteq accept[r]\}| \geq n - t$ ;
  3. **step 2**  $send(r, 2, accept[r])$  to all; */\* the set  $accept[r]$  is being continuously updated in the background \*/*
  4. **wait until**  $|\{j \mid received(r, 2, \pi_j) \text{ from } p_j, \text{ and } \pi_j \subseteq accept[r]\}| \geq n - t$ ;
  5. **return**  $accept[r]$ .
- 

complete the first round of COMMONCORE, and 0 otherwise. Observe that if 1 appears in entry  $T[i, j]$ , the  $accept_i[r]$  sent by  $p_i$  in the second round contains all the  $n - t$  IDs appearing in the initial  $accept_j[r]$  sent by  $p_j$  in the first round of the COMMONCORE protocol.

Since all correct processors will eventually invoke COMMONCORE,  $T$  will contain at least  $(n - t)(t + 1)$  entries with 1. This implies that there is an ID of a correct processor, say  $\bar{p}$ , that appears in at least  $t + 1$  rows. Thus, there are at least  $t + 1$  correct processors whose second round set includes the  $n - t$  IDs that appear in the initial set of  $\bar{p}$ . Before completing the protocol, each processor waits to get the sets of  $n - t$  processors, so it includes the set of at least one of these  $t + 1$  processors, thus includes the set of  $n - t$  IDs appearing in the initial list of  $\bar{p}$ .

**Lemma 3.** *For  $n > 3t$ , the protocol presented in Algorithm 4 implements the COMMONCORE properties.*

To obtain the final protocol we add the COMMONCORE invocation to the simulation protocol presented in Algorithm 1. We invoke the COMMONCORE protocol on all the  $accept$  sets of a given round after completing Line 7 and before executing Line 10 of Algorithm 1. The output of the COMMONCORE is used in Line 10 as the set of processors from which we received messages from in that round. Algorithm 5 presents the complete protocol.

**Theorem 1.** *Given a deterministic protocol  $P$  that is viewed as a function  $\mathcal{F}(M_{r-1}, S_p, r)$ , for  $r \geq 1$ , in a BI-MO( $B_{initial}$ ) system, the protocol presented in Algorithm 5 simulates it in a  $t$ -Byzantine resilient asynchronous system, provided that  $n > 3t$ .*

**Corollary 1.** *For  $n > 3t$  and deterministic protocols,  $t$ -Byzantine resilient asynchronous system and BI-MO( $B_{initial}$ ) system are equivalent.*

**Theorem 2.** *For  $n > 3t$  and deterministic protocols,  $t$ -Byzantine resilient asynchronous system is equivalent to  $t$ -resilient asynchronous system in a Cuckoo model (i.e., in which the adversary only replaces the inputs to some  $t$  processors).*

For the synchronous model with a fixed set of faults we obtain a similar result. To obtain the result we do not need the COMMONCORE part and Algorithm 1 is enough.<sup>2</sup>

**Theorem 3.** *For  $n > 3t$  and deterministic protocols. A synchronous Byzantine fault system is equivalent to a synchronous system in which the adversary replaces the inputs to some  $t$  processors and outgoing messages from them it may delete in the first round, and in any future round it may delete outgoing messages from some set of  $t$  processors.*

---

<sup>2</sup>One can somewhat optimize the protocols in the synchronous case, but conceptually the simulation remains the same.



---

**Algorithm 5:** Simulating a BI-MO( $B_{initial}$ ) system deterministic protocol  
in an asynchronous system that satisfies with  $n > 3t$ .

---

```

1. set  $\forall k \text{ } accept[k] := \emptyset;$  /* the sets of accepted senders at the given rounds; executed at processor  $p$  */
2. set  $\mathcal{M} := \emptyset; \bar{\mathcal{M}} := \emptyset;$  /* the set of accepted messages that were not processed yet, and the set of processed messages */
3. set  $O := \emptyset;$  /* the set of processors that their SM reached an output at the processor */

4.  $r := 1;$  /* the round number */
5. invoke CO_SEND( $r, p$ ) to broadcast  $\mathcal{I};$  /* broadcast the input value, a processor sends also to itself */

6. do until  $SM_p$  halts: /* participate in all CO_SEND( $\ell, *$ ),  $\ell \leq r$ , protocols */
7.   wait until  $|accept[r]| \geq n - t$  and  $p \in accept[r];$ 
8.    $accept[r] := \text{COMMONCORE}(accept[r]);$  /* the 2 rounds protocol to converge to shared  $n - t$  */
9.    $r := r + 1;$ 
10.  invoke CO_SEND( $r, p$ ) to broadcast  $accept[r - 1];$  /* broadcast the accepted and processed set in round  $r - 1$  */
11. end.

```

**In the Background:**

```

12. Execute for each  $\langle r', p_i, \pi \rangle \in \mathcal{M};$  /* accepted message from  $p_i$  for round  $r'$  */
13.   if  $r' = 1$  then start  $SM_i$  with input  $\pi;$  /* start a SM with the initial input */
14.   if  $r' > 1$  then
15.     let  $M := \{m_j \mid p_j \in \pi \text{ and } SM_j[r' - 1] \text{ sends } m_j \text{ to } p_i\};$  /* the values  $p_i$  should have received */
16.      $SM_i[r'] := \mathcal{F}(M, SM_i[r' - 1], r');$  /* apply protocol  $\mathcal{F}$  to determine the next state of  $SM_i$  */
17.     if  $SM_i$  outputs then  $O := O \cup \{p_i\};$  /* Denote the outputting processor */
18.    $\mathcal{M} := \mathcal{M} \setminus \langle r', p_i, \pi \rangle;$ 
19.    $\bar{\mathcal{M}} := \bar{\mathcal{M}} \cup \langle r', p_i \rangle;$ 
20.    $accept[r'] := accept[r'] \cup \{p_i\}.$ 

```

---

## 6 Reliable Broadcast

In this section we present the traditional Reliable Broadcast protocol using the generalized faulty set convention.

**Definition 1** (Collection of Sets of Potentially Faulty Processors One of which can be Chosen by the Adversary). *The set  $\mathcal{B}$ , called the collection of bad-sets, is closed under inclusion, i.e. if  $B \in \mathcal{B}$  and  $B' \subseteq B$ , then  $B' \in \mathcal{B}$ . Let  $\mathcal{G}$  be  $\{\Pi \setminus B \mid B \in \mathcal{B}\}$ .*

*B1:  $\mathcal{B}$  satisfies the Benign Fault predicate if: any 2 potentially bad sets of processors  $B_i, B_j \in \mathcal{B}$ , satisfy  $\Pi \not\subseteq B_i \cup B_j$ .*

*B2:  $\mathcal{B}$  satisfies the Byzantine Fault predicate if: any 3 potentially bad sets  $B_i, B_j, B_k \in \mathcal{B}$ , satisfy  $\Pi \not\subseteq B_i \cup B_j \cup B_k$ .*

Thus, the only difference between the Benign Fault predicate and the Byzantine Fault predicate is the number of sets that do not cover the entire set of processors.

**Corollary 2.** *The definition of the potentially Faulty sets that satisfy the Byzantine Fault predicate implies:*

*G1: For every  $G \in \mathcal{G}$ , and any  $B_i, B_j \in \mathcal{B}$ , it holds that  $G \not\subseteq B_i \cup B_j$ .*

*G2: For any 2 potentially good sets  $G_i, G_j \in \mathcal{G}$ , it holds that  $G_i \cap G_j \subsetneq B, B \in \mathcal{B}$ .*

*Proof.* Proving G1: If G1 does not hold then  $\Pi \subseteq B_i \cup B_j \cup \{\Pi \setminus G\}$ , a contradiction.

Proving G2: If G2 does not hold then  $\Pi \subseteq \{\Pi \setminus G_i\} \cup \{\Pi \setminus G_j\} \cup \{G_i \cap G_j\}$ , a contradiction, since all sets belong to  $\mathcal{B}$ .  $\square$

The Reliable Broadcast properties are:

RB1 If the sender  $s$  is correct when it invokes RB with a value  $v$ , then every processor will eventually accept  $\text{RB}(s)$  with the value  $v$ .

RB2 If any (correct) processor accepts  $\text{RB}(s)$  with a value  $v'$ , then every processor will accept it with the same value.

RB3 If sender  $s$  did not invoke  $\text{RB}(s)$  no processor will accept  $\text{RB}(s)$ .

Notice that in our model there is no need to use the word “correct” in stating RB2.

---

**Algorithm 6:**  $\text{RB}(k, s)$ : Reliable Broadcast

with asynchronous system that satisfies the Byzantine Fault predicate

---

- |    |                                                                                                                                                                                     |                                                                                                  |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
|    |                                                                                                                                                                                     | /* executed by processor $p$ with sender $s$ , invoked once per $k$ per sender */                |
| 1. | <b>set</b> $V := \emptyset$ ;                                                                                                                                                       | /* the set of $m_1$ and $m_2$ messages received, each processor also sends messages to itself */ |
| 2. | <b>Init:</b> if $p = s$ <b>then</b> send $v_s$ to all;                                                                                                                              | /* $s$ is the sender and $v_s$ the value it broadcasts */                                        |
| 3. | <b>Upon receiving a protocol message:</b>                                                                                                                                           |                                                                                                  |
| 4. | <b>case</b> received $v$ from $s$ : send $m_1(v)$ to all;                                                                                                                           | /* executed at most once per protocol invocation */                                              |
| 5. | <b>case</b> received $m$ from $q$ : add $m$ to $V$ ;                                                                                                                                | /* add this $m_1$ or $m_2$ message to $V$ */                                                     |
| 6. | <b>case</b> $V$ contains $m_1(v)$ from a set $G, G \in \mathcal{G}$ <b>or</b> $m_2(v)$ from a set $G', G' \not\subseteq B$ , for any $B \in \mathcal{B}$ :<br>send $m_2(v)$ to all; | /* send at most once per protocol invocation */                                                  |
| 7. | <b>case</b> $V$ contains $m_2(v)$ from a set $G \in \mathcal{G}$ :                                                                                                                  | /* process the sender's message */                                                               |
| 8. | <b>accept</b> $\text{RB}(k, s)$ with value $v$ .                                                                                                                                    | /* accept message $v$ as the message sent by processor $s$ with index $k$ */                     |
- 

The Reliable Broadcast protocol, [Algorithm 6](#), is invoked per processor per an index  $k$  per sending round and consists of 4 conceptual steps. Initially (step 1, [Line 2](#)) the sender of the current instance of the protocol sends its initial value to everyone. Thus, everyone should wait to receive the appropriate initial value. Due to asynchrony it may take time, but without faults, it would eventually arrive to everyone. Because of maliciousness, the message may not arrive to every processor. Moreover, conflicting values might be sent to different processors. The following steps intend to address exactly these difficulties.

If a processor receives an initial value (step 2, [Line 4](#)) it notifies every processor by sending  $m_1(v)$  message. Malicious behavior may cause different processors to send  $m_1$  messages for different values. Each processor sends at most a single  $m_1$  message per invocation of the protocol (per round). A processor may receive several  $m_1$  messages, even if it did not receive an initial value.

In step 3 ([Line 6](#)), a processor that has received identical copies of  $m_1$  messages (*i.e.*, for the same value) from a set  $G \in \mathcal{G}$ , sends an  $m_2$  message. Notice that if the original sender is correct, this will eventually happen at every correct processor. Observe that by [Corollary 2](#) no two correct processors send  $m_2$  messages with conflicting values, since the protocol instructs a correct processor to send at most a single  $m_1$  message, and no two correct processors will get identical copies of  $m_1$  messages for different values from different set in  $\mathcal{G}$ . Notice that a processor may receive several  $m_2$  messages without receiving enough identical copies of  $m_1$  messages from a set  $G \in \mathcal{G}$ . If it receives  $m_2$  identical messages from a set  $G', G' \not\subseteq B$ , for any  $B \in \mathcal{B}$ , it knows that at least one correct processor have sent one, so it can also join in by sending an  $m_2$  message (potentially skipping step 2 on the way).

To complete the protocol (step 4, [Line 7](#)) a processor waits to receive  $m_2$  identical messages from a set  $G \in \mathcal{G}$ . Once it receives that many identical  $m_2$  messages, and since any correct processor will either send the identical message or none, it knows that eventually every processor will receive the  $m_2$  message from at least a set  $G'$ ,  $G' \not\subseteq B$ , for any  $B \in \mathcal{B}$ . Any such correct processor will send an  $m_2$  message to everyone else, which leads to everyone eventually receiving identical  $m_2$  messages from a set  $G \in \mathcal{G}$ . Thus, if any correct processor will agree to accept the message, eventually every correct processor will accept and process all the prior messages and will accept the message.

Notice that the way [Line 6](#) is presented it implies scanning an exponential number of sets - but it is clear that it is enough to consider maximal sets, and the complexity becomes the number of maximal sets.

## 7 Related Work

The attempts of reducing Byzantine to Benign have a long history. Back in 1988 Coan [13] considered running an asynchronous protocol written for the benign setting on a machine in which processors might fail in a Byzantine manner. To deal with inputs he assumed that the inputs satisfy some predicate, and then required the protocol to be written in a special form. This allowed him to check “backward compatibility” and discard incorrect messages. But, it was not clear whether any asynchronous protocol can be put in the form he required.

We, here, use the same idea as that of Coan, only that we do not require the special protocol form required by Coan. We check “backward compatibility” just by asking a processor to send the set of processors it read from. Since it is sent via Reliable Broadcast it allows the rest of the processors to delay accepting the message until they simulate locally all the processors claimed in the message and then derive the interpreted value themselves. In hindsight we believe Coan method can be tweaked to ours. Ours might be just an inductive full-information version of Coan ideas.

As for general results reducing the Asynchronous Byzantine to the Benign such a reduction was derived recently in [27], and rediscovered in [14]. But both reductions were explained only for *colorless* tasks. They might apply to all tasks, but we will speculate that the fact that they could not ascribe meaning to output of “byzantine processors” held them back. In fact, some of the author of [27] were involved in formulating the beautiful task of Multi-Dimensions Vector  $\epsilon$ -agreement [26]. The task is motivated by the presence of Byzantine processors, but its computability is possible to analyze in the Benign model according to the Cuckoo model. Nevertheless its computability in [26] is handled in the Byzantine setting with all the complications and clutter it introduces. Reference [27] is mute about that possibility, although we view it as a “killer application” to the benefit of the Cuckoo Model.

Thus the central contribution of our paper is conceptual. By making all processors “correct” and allow them to join the computation once the Byzantine attack ceases we are forced to think a complete analogy between the Byzantine and the Benign modulo changed inputs. The “byzantine processors” cluttered the view of researchers, hence for instance, we are the first to notice that the Asynchronous Adversary model of 2009 [16] applies verbatim to the Byzantine model. Missing the “correct” model might have cause researchers to miss results which are essentially in plain view.

In between Coan [13] and [27] there were several attempts to simplify the  $t$  resilient asynchronous Byzantine model through ideas of simulating simpler models. Attiya and Welch [7] reduced the problem to Identical Byzantine. The pioneering work of Bracha [11, 12] was focused on improving the probabilistic protocol of Ben-Or [9] from  $n/5$  to  $n/3$  and in order to do so Bracha developed a basic tool to limit the power of the Byzantine adversary, The simulation we introduce in the paper makes use of this tool as part

of the building block we introduce. Srikanth and Toueg [31] considered simulating the power of a signature scheme to limit the Byzantine adversary, both in a synchronous system and an asynchronous one. Neiger and Toueg [28] introduced direct simulations between models in order to solve consensus, but their simulations are limited to synchronous models.

Last but not least as mentioned in the introduction our motivating papers were [5] and [20, 15]. Reference [20] implies that processors  $t$ -resiliently jointly can march any number (greater than  $t$ ) of state machines forward with at most  $t$  not progressing. In [15] state-machines are considered to be read-write threads (and the commands to be what a processors proposes a “read” value based on its internal simulation), hence a protocol. Hence it makes an execution a group effort - our main motivating observation as to how this might apply to the Byzantine. The [20] is a generalization of Schneider [30] using a state machine for implementing fault-tolerant services. It generalizes [30] from when consensus is available to the case when set-consensus is available rather than consensus. The  $t$  resilience allows for  $k + 1$  set-consensus.

## 8 Conclusions

We presented a new view on the Byzantine model. In this view processors are not taken over by a malicious Adversary. The Adversary takes over the outgoing communication attempted by some processors. For all practical purposes the observable behavior is the same. Yet theoretically it allows us to consider “reviving” a processor after an attack ceases as throughout the execution it snooped on the system progress. Thus, this raised the challenge as to whether there is a way to completely reduce the Byzantine to the Benign with no qualifications.

This paper present a way to do this. It is not prefect. We assume that faulty behavior will not be experienced further at some processor without output, *immediately* after a new output is delivered past  $n - t - 1$  outputs. Perfection would mean to replace this assumption of “immediately” with “eventually.” We leave it as an open question whether this can be done (e.g. by changing the assumption that the adversary can remove a message, to the one where the adversary can delay a message ad infinitum). Recovery necessitates that a processor trying revival will resend a message. We discarded various schemes of “eventual” as we could not bound the number of these extra resend messages.

Yet, with this somewhat flawed assumption, that nevertheless does not affects “observable behavior,” we are able to reduce the extreme of Asynchronous Byzantine, to the other extreme of the Synchronous Benign. What is left is to show that such reductions are true for other models e.g. those that utilize objects in their communications. Indeed, implicitly such a reduction has been shown in [19] for objects of set-consensus types, since there, objects are replaced by a restriction on runs.

It is now about making the reduction in this paper efficient, and finding “killer applications,” beyond [25, 32, 26]. Another direction is turning a probabilistic asynchronous consensus algorithms, into probabilistic algorithms where the faults are benign.

The emulations discussed in this paper assume deterministic protocol. In order to extend the emulations above to randomized protocols, one can’t just ask a processor to distribute together with its collected set its random choice for the current round. The reason is that the adversary can make the Byzantine processors to not draw the random bits from the expected distribution. To deal with that all processors collectively can choose the random choices for each processor once it broadcasts its candidates’ set. In case  $n > 4t$  one can compute any probabilistic function using Asynchronous Multi-Party-Computation, in the presence of up to  $t$  Byzantine faults, see [10, 8]. One can use that as a building block in our protocols. The question of doing that for  $n > 3t$  is an open problem.

## References

- [1] The Book of Ecclesiastes, 1:9.
- [2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.
- [3] Ittai Abraham, Marcos K. Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing: 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, pages 4–19, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [4] Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In Teruo Higashino, editor, *Principles of Distributed Systems: 8th International Conference, OPODIS 2004, Grenoble, France, December 15-17, 2004, Revised Selected Papers*, pages 229–239, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [5] Yehuda Afek and Eli Gafni. A simple characterization of asynchronous computations. *Theor. Comput. Sci.*, 561:88–95, 2015.
- [6] Bernd Altmann, Matthias Fitzi, and Ueli M. Maurer. Byzantine agreement secure against general adversaries in the dual failure model. In Prasad Jayanti, editor, *Distributed Computing, 13th International Symposium, Bratislava, Slavak Republic, September 27-29, 1999, Proceedings*, volume 1693 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1999.
- [7] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.
- [8] Zuzana Beerliová-Trubíniová and Martin Hirt. *Simple and Efficient Perfectly-Secure Asynchronous MPC*, pages 376–392. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [9] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983.
- [10] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, STOC '93*, pages 52–61, New York, NY, USA, 1993. ACM.
- [11] Gabriel Bracha. An asynchronous  $[(n - 1)/3]$ -resilient consensus protocol. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, PODC '84*, pages 154–162, New York, NY, USA, 1984. ACM.
- [12] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130 – 143, 1987.
- [13] B. A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, Dec 1988.
- [14] Imbs Damien, Raynal Michel, and Stainer Julien. Are byzantine failures really different from crash failures? In *DISC 16*, pages 215–229, 2016.

- [15] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov. Wait-freedom with advice. *Distributed Computing*, 28(1):3–19, 2015.
- [16] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3):137–147, 2011.
- [17] D. Dolev, N. A. Lynch, E. Stark, W. E. Weihl, and S. Pinter. Reaching approximate agreement in the presence of faults. *J. of the ACM*, 33:499–516, 1986.
- [18] Danny Dolev and Eli Gafni. Some garbage in - some garbage out: Asynchronous t-byzantine as asynchronous benign t-resilient system with fixed t-trojan-horse inputs. *CoRR*, abs/1607.01210, 2016.
- [19] Petr Kuznetsov Eli Gafni, Yuan He and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. In *Proc. of 20th Int. Conference on Principles of Distributed Systems (OPODIS’03)*, 2016.
- [20] Eli Gafni and Rachid Guerraoui. Generalized universality. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 17–27. Springer, 2011.
- [21] Eli Gafni and Petr Kuznetsov. Relating  $L$ -resilience and wait-freedom via hitting sets. In Marcos Kawazoe Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit Roy Choudhury, editors, *Distributed Computing and Networking - 12th International Conference, ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings*, volume 6522 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2011.
- [22] Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 105–113, 2010.
- [23] Martin Hirt and Ueli M. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 25–34. ACM, 1997.
- [24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–301, 1982.
- [25] Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC ’13, pages 391–400, New York, NY, USA, 2013. ACM.
- [26] Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28(6):423–441, 2015.
- [27] Hammurabi Mendes, Christine Tasson, and Maurice Herlihy. Distributed computability in byzantine asynchronous systems. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC ’14, pages 704–713, New York, NY, USA, 2014. ACM.

- [28] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 248–262, New York, NY, USA, 1988.
- [29] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr 1980.
- [30] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- [31] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.
- [32] Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 65–73, New York, NY, USA, 2013. ACM.